react-webpack-cookbook

hainuo

Published
with GitBook

目录

1. 介绍
2. 介绍 Webpack
    i. 第一步
    ii. 开始工作流
    iii. 浏览器自动刷新
    iv. 引入文件
3. React JS
    i. 配置 React JS
    ii. 优化重合并
    iii. Flow
4. CSS, Fonts and Images
    i. 加载 CSS
    ii. 自动刷新 CSS
    iii. 加载 LESS 和 SASS
    iv. 内联 images
    v. 内联 fonts
5. 部署策略
    i. 发布配置
    ii. 合并成单文件
    iii. 分离应用和第三方
    iv. 多重入口
    v. 懒加载入口文件
    vi. 同构渲染
6. 进阶
    i. 优化开发
    ii. 热加载组件
    iii. 使用下一代 JavaScript
    iv. 优化缓存
    v. 匹配器
    vi. 懒加载入口
    vii. 创建一个复用代码
    viii. 理解 Chunks
    ix. 创建库

React 和 Webpack 小书 - 一本介绍在 React JS 中使用 Webpack 的小书。

> 中文版

贡献

若书中有遗漏或者错误的地方，可以按照下面的方式：

1. Fork 这个仓库
2. 切换到 zh-cn 分支
3. 在 `/content` 里做修改
4. 提交一个 PR

或者可以到原仓库中 提 issue。

注意 `gh-pages` 内容是自动生成的。

Gitbook 生成器

生成器会自动生成 wiki 和网站，为了能够让它提交 `gh-pages` 分支，这样使用它：

1. `npm install`
2. `npm run generate-gitbook`

它会自动生成 `/gh-pages`，你可以直接通过文件系统来访问。

如果要发布内容，那么输入 `npm run deploy-gitbook`，它会自动更新 `gh-pages` 分支。

In web development we deal with a lot of small technical artifacts. You use HTML to describe page structure, CSS how to style it and JavaScript for logic. Or you can replace HTML with something like Jade, CSS with Sass or LESS, JavaScript with CoffeeScript, TypeScript and the ilk. In addition you have to deal with project dependencies (ie. external libraries and such).

在 Web 开发历程上，我们构建了很多小型的技术解决方案，比如用 HTML 去描述页面结构，CSS 去描述页面样式，JavaScript 去描述页面逻辑，或者你也可以用一些比如 Jade 去取代 HTML，用 Sass 或 Less 去取代CSS，用 CoffeeScript 或者 TypeScript 之类的去取代 JavaScript，不过项目中的依赖可能是一件比较烦恼的事情。（需要安装额外很多的库）

There are good reasons why we use these various technologies. Regardless of what we use, however, we still want to end up with something that can be run on the browsers of the clients. This is where build systems come in. Historically speaking there have been many. Make is perhaps the most known one and still a viable option in many cases. In the world of frontend development particularly Grunt and Gulp have gained popularity. Both are made powerful by plugins. NPM, the Node.js package manager, is full of those.

这里有很多为什么我们需要尝试那些新技术的理由。不管我们用什么，总之，我们还是希望使用那些能够处理在浏览器端的方案，所以出来了编译方案。历史上已经有很多分享了，比如 Make 可能是很多解决方案中最知名且是可行的方案。Grunt 和 Gulp 是在是前端的世界中最流行的解决方案，他们两个都有很多非常有用的插件。NPM（Node.js 的包管理器）则包含了他们两个。

Grunt

Grunt is the older project. It relies on plugin specific configuration. This is fine up to a point but believe me, you don't want to end up having to maintain a 300 line `Gruntfile`. The approach simply turns against itself at some point. Just in case you are curious what the configuration looks like, here's an example from Grunt documentation:

Grunt 是相比后面几个更早的项目，他依赖于各种插件的配置。这是一个很好的解决方案，但是请相信我，你不会想看到一个 300 行的 `Gruntfile`。如果你好奇 Grunt 的配置会如何，那么这里是有个从 Grunt 文档 的例子：

```
module.exports = function(grunt) {

  grunt.initConfig({
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');

  grunt.registerTask('default', ['jshint']);

};
```

Gulp

Gulp takes a different approach. Instead of relying on configuration per plugin you deal with actual code. Gulp builds on top of the tried and true concept of piping. If you are familiar with Unix, it's the same here. You simply have sources, filters and sinks. In this case sources happen to match to some files, filters perform some operations on those (ie. convert to

JavaScript) and then output to sinks (your build directory etc.). Here's a sample `Gulpfile` to give you a better idea of the approach taken from the project README and abbreviated a bit:

Gulp 提供了一个不一样的解决方案，而不是依赖于各种插件的配置。Gulp 使用了一个文件流的概念。如果你熟悉 Unix，那么 Gulp 对你来说会差不多，Gulp 会提供你一些简单化的操作。在这个解决方案中，是去匹配一些文件然后操作（就是说和 JavaScript 相反）然后输出结果（比如输出在你设置的编译路径等）。这里有一个简单的 `Gulpfile` 的例子：

```
var gulp = require('gulp');
var coffee = require('gulp-coffee');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var del = require('del');

var paths = {
  scripts: ['client/js/**/*.coffee', '!client/external/**/*.coffee'],
};

// 不是所有的任务需要使用 streams
// 一个 gulpfile 只是另一个node的程序，所以你可以使用所有 npm 的包
gulp.task('clean', function(cb) {
  // 你可以用 `gulp.src` 来使用多重通配符模式
  del(['build'], cb);
});

gulp.task('scripts', ['clean'], function() {
  // 压缩和复制所有 JavaScript （除了第三方库）
  // 加上 sourcemaps
  return gulp.src(paths.scripts)
    .pipe(sourcemaps.init())
      .pipe(coffee())
      .pipe(uglify())
      .pipe(concat('all.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'));
});

// 监听文件修改
gulp.task('watch', function() {
  gulp.watch(paths.scripts, ['scripts']);
});

// 默认任务（就是你在命令行输入 `gulp` 时运行）
gulp.task('default', ['watch', 'scripts']);
```

Given the configuration is code you can always just hack it if you run into troubles. You can wrap existing Node.js modules as Gulp plugins and so on. You still end up writing a lot of boilerplate for casual tasks, though.

这些配置都是代码，所以当你遇到问题也可以修改，你也可以使用已经存在的 Gulp 插件，但是你还是需要写一堆模板任务。

Browserify

Dealing with JavaScript modules has always been a bit of a problem given the language actually doesn't have a concept of module till ES6. Ergo we are stuck with the 90s when it comes to browser environment. Various solutions, including AMD, have been proposed. In practice it can be useful just to use CommonJS, the Node.js format, and let tooling deal with the rest. The advantage is that you can often hook into NPM and avoid reinventing the wheel.

处理 JavaScript 模块一直是一个大问题，因为这个语言在 ES6 之前没有这方面的概念。因此我们还是停留在90年代，各种解决方案，比如提出了 AMD。在实践中只使用 CommonJS （ Node.js 所采用的格式）会比较有帮助，而让工具去处理剩下的事情。它的优势是你可以发布到 NPM 上来避免重新发明轮子。

Browserify solves this problem. It provides a way to bundle CommonJS modules together. You can hook it up with Gulp. In

addition there are tons of smaller transformation tools that allow you to move beyond the basic usage (ie. watchify provides a file watcher that creates bundles for you during development automatically). This will save some effort and no doubt is a good solution up to a point.

Browserify 解决了这个问题，它提供了一种可以把模块集合到一起的方式。你可以用 Gulp 调用它，此外有很多转换小工具可以让你更兼容的使用（比如 watchify 提供了一个文件监视器帮助你在开发过程中更加自动化地把文件合并起来），这样会省下很多精力。毋庸置疑，一定程度来讲，这是一个很好的解决方案。

Webpack

Webpack expands on the idea of hooking into CommonJS `require`. What if you could just `require` whatever you needed in your code, be it CoffeeScript, Sass, Markdown or something? Well, Webpack does just this. It takes your dependencies, puts them through loaders and outputs browser compatible static assets. All of this is based on configuration. Here is a sample configuration from the official Webpack tutorial:

Webpack 扩展了 CommonJs 的 `require` 的想法，比如你想在 CoffeeScript、Sass、Markdown 或者其他什么代码中 `require` 你想要的任何代码的话？那么 Webpack 正是做这方面的工作。它会通过配置来取出代码中的依赖，然后把他们通过加载器把代码兼容地输出到静态资源中。这里是一个 Webpack 官网 上的例子：

```
module.exports = {
    entry: "./entry.js",
    output: {
        path: __dirname,
        filename: "bundle.js"
    },
    module: {
        loaders: [
            { test: /\.css$/, loader: "style!css" }
        ]
    }
};
```

In the following sections we'll build on top of this idea and show how powerful it is. You can, and probably should, use Webpack with some other tools. It won't solve everything. It does solve the difficult problem of bundling, however, and that's one worry less during development.

在接下来的章节中我们会使用 Webpack 来构建项目来展示它的能力。你可以用其他工具和 Webpack 一起使用。它不会解决所有事情，只是解决一个打包的难题，无论如何，这是在开发过程中需要解决的问题。

> Before getting started you should make sure you have a recent version of Node.js and NPM installed. See nodejs.org for installation details. We'll use NPM to set up various tools.
>
> 在开始之前，你需要把你的 Node.js 和 NPM 都更新到最新的版本。访问 nodejs.org 查看安装详情。我们将会使用 NPM 安装一些工具。

Getting started with Webpack is straightforward. I'll show you how to set up a simple project based on it. As a first step, set a directory for your project and hit `npm init` and fill in some answers. That will create a `package.json` for you. Don't worry if some fields don't look ok, you can modify those later.

开始使用 Webpack 非常简单，我会展示给你看使用它的一个简单的项目。第一步，为你的项目新建一个文件夹，然后输入 `npm init`，然后填写相关问题。这样会为你创建了 `package.json`，不用担心填错，你可以之后修改它。

安装 Webpack

Next you should get Webpack installed. We'll do a local install and save it as a project dependency. This way you can invoke the build anywhere (build server, whatnot). Run `npm i webpack --save-dev`. If you want to run the tool, hit `node_modules/.bin/webpack`.

接下来我们安装 Webpack，我们要把它安装在本地，然后把它作为项目依赖保存下来。这样你可以在任何地方编译（服务端编译之类的）。输入 `npm i webpack --save-dev`。如果你想运行它，就输入 `node_modules/.bin/webpack`。

目录结构

Structure your project like this:

项目的目录结构长这样：

- /app
  - main.js
  - component.js
- /build
  - bundle.js (自动创建)
  - index.html
- package.json
- webpack.config.js

In this case we'll create `bundle.js` using Webpack based on our `/app`. To make this possible, let's set up `webpack.config.js`.

我们会使用 Webpack 在我们的 `/app` 里来自动创建 `bundle.js` 。接下来，我们来设置 `webpack.config.js`。

设置 Webpack

Webpack 的配置文件长这样：

webpack.config.js

```
var path = require('path');


module.exports = {
    entry: path.resolve(__dirname, 'app/main.js'),
```

```
    output: {
        path: path.resolve(__dirname, 'build'),
        filename: 'bundle.js',
    },
};
```

运行你的第一个编译

Now that we have basic configuration in place, we'll need something to build. Let's start with a classic `Hello World` type of app. Set up `/app` like this:

现在我们有了一个最简单的配置，我们需要有什么东西去编译，让我们开始一个经典的 `Hello World`，设置 `/app` 像这样：

app/component.js

```
'use strict';

module.exports = function () {
    var element = document.createElement('h1');

    element.innerHTML = 'Hello world';

    return element;
};
```

app/main.js

```
'use strict';
var component = require('./component.js');

document.body.appendChild(component());
```

Now run `webpack` in your terminal and your application will be built. A *bundle.js* file will appear in your `/build` folder. Your *index.html* file in the `build/` folder will need to load up the application.

现在在你的命令行运行 `webpack`，然后你的应用会开始编译，一个 `bundle.js` 文件就这样出现在你的 `/build` 文件夹下，需要在 `build/` 下的 *index.html* 去启动项目。

build/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

> It would be possible to generate this file with Webpack using html-webpack-plugin. You can give it a go if you are feeling adventurous. It is mostly a matter of configuration. Generally this is the way you work with Webpack.
>
> 这个文件可以用 html-webpack-plugin 来生成。如果你觉得冒险，那就把剩下的工具交给它来做。使用它就只有一个配置的问题。一般来说使用 Webpack 来工作就是这么个套路。

运行应用

Just double-click the *index.html* file or set up a web server pointing to the `build/` folder.

只要双击 *index.html* 或者设置一个 Web 服务指向 `build/` 文件夹。

设置 `package.json` *scripts*

It can be useful to run build, serve and such commands through `npm`. That way you don't have to worry about the technology used in the project. You just invoke the commands. This can be achieved easily by setting up a `scripts` section to `package.json`.

`npm` 是一个非常好用的用来编译的指令，通过 `npm` 你可以不用去担心项目中使用了什么技术，你只要调用这个指令就可以了，只要你在 `package.json` 中设置 `scripts` 的值就可以了。

In this case we can move the build step behind `npm run build` like this:

在这个案例中我们把编译步骤放到 `npm run build` 中是这样：

1. `npm i webpack --save` - If you want to install Webpack just a development dependency, you can use `--save-dev`. This is handy if you are developing a library and don't want it to depend on the tool (bad idea!).
2. Add the following to `package.json`:

1. `npm i webpack --save` - 如果你想要把 Webpack 作为一个项目的开发依赖，就可以使用 `--save-dev`，这样就非常方便地让你在开发一个库的时候，不会依赖工具（但不是个好方法！）。
2. 把下面的内容添加到 `package.json` 中。

```
"scripts": {
  "build": "webpack"
}
```

To invoke a build, you can hit `npm run build` now.

现在你可以输入 `npm run build` 就可以编译了。

Later on this approach will become more powerful as project complexity grows. You can hide the complexity within `scripts` while keeping the interface simple.

当项目越发复杂的时候，这样的方法会变得越来越有效。你可以把所有复杂的操作隐藏在 `scripts` 里面来保证界面的简洁。

The potential problem with this approach is that it can tie you to a Unix environment in case you use environment specific commands. If so, you may want to consider using something environment agnostic, such as gulp-webpack.

不过潜在的问题是这种方法会导致如果你使用一些特殊的指令的时候只能在 Unix 环境中使用。所以如果你需要考虑一些未知的环境中的话，那么 gulp-webpack 会是一个好的解决方案。

> Note that NPM will find Webpack. `npm run` adds it to the `PATH` temporarily so our simple incantation will work.
>
> 注意 NPM 会找到 Webpack，`npm run` 会把他临时加到 `PATH` 来让我们这个神奇的命令工作。

Hitting `npm run build` all the time will get boring eventually. Fortunately we can work around that quite easily. Let's set up `webpack-dev-server`.

如果需要一直输入 `npm run build` 确实是一件非常无聊的事情，幸运的是，我们可以把让他安静的运行，让我们设置 `webpack-dev-server`。

设置 `webpack-dev-server`

As a first step, hit `npm i webpack-dev-server --save`. In addition we'll need to tweak `package.json` *scripts* section to include it. Here's the basic idea:

第一步，输入 `npm i webpack-dev-server --save`，此外，我们需要去调整 `package.json` *scripts* 部分去包含这个指令，下面是基本的设置：

package.json

```
{
  "scripts": {
    "build": "webpack",
    "dev": "webpack-dev-server --devtool eval --progress --colors --hot --content-base build"
  }
}
```

When you run `npm run dev` from your terminal it will execute the command stated as a value on the **dev** property. This is what it does:

当你在命令行里运行 `npm run dev` 的时候他会执行 **dev** 属性里的值。这是这些指令的意思：

1. `webpack-dev-server` - Starts a web service on localhost:8080
2. `--devtool eval` - Creates source urls for your code. Making you able to pinpoint by filename and line number where any errors are thrown
3. `--progress` - Will show progress of bundling your application
4. `--colors` - Yay, colors in the terminal!
5. `--content-base build` - Points to the output directory configured

1. `webpack-dev-server` - 在 localhost:8080 建立一个 Web 服务器
2. `--devtool eval` - 为你的代码创建源地址。当有任何报错的时候可以让你更加精确地定位到文件和行号
3. `--progress` - 显示合并代码进度
4. `--colors` - Yay，命令行中显示颜色！
5. `--content-base build` - 指向设置的输出目录

To recap, when you run `npm run dev` this will fire up the webservice, watch for file changes and automatically rebundle your application when any file changes occur. How neat is that!

总的来说，当你运行 `npm run dev` 的时候，会启动一个 Web 服务器，然后监听文件修改，然后自动重新合并你的代码。真的非常简洁！

Go to **http://localhost:8080** and you should see something.

访问 **http://localhost:8080** 你会看到效果。

When **webpack-dev-server** is running it will watch your files for changes. When that happens it rebundles your project and notifies browsers listening to refresh. To trigger this behavior you need to change your *index.html* file in the `build/` folder.

当运行 **webpack-dev-server** 的时候，它会监听你的文件修改。当项目重新合并之后，会通知浏览器刷新。为了能够触发这样的行为，你需要把你的 *index.html* 放到 `build/` 文件夹下，然后做这样的修改：

build/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <script src="http://localhost:8080/webpack-dev-server.js"></script>
    <script src="bundle.js"></script>
  </body>
</html>
```

We added a script that refreshes the application when a change occurs. You will also need to add an entry point to your configuration:

我们需要增加一个脚本当发生改动的时候去自动刷新应用，你需要在配置中增加一个入口点。

```
var path = require('path');


module.exports = {
    entry: ['webpack/hot/dev-server', path.resolve(__dirname, 'app/main.js')],
    output: {
        path: path.resolve(__dirname, 'build'),
        filename: 'bundle.js',
    },
};
```

Thats it! Now your application will automatically refresh on file changes.

就是这样！现在你的应用就可以在文件修改之后自动刷新了。

默认环境

In the example above we created our own *index.html* file to give more freedom and control. It is also possible to run the application from **http://localhost:8080/webpack-dev-server/bundle**. This will fire up a default *index.html* file that you do not control. It also fires this file up in an iFrame allowing for a status bar to indicate the status of the rebundling process.

在上面的例子中我们创建了 *index.html* 文件来获取更多的自由和控制。同样也可以从 **http://localhost:8080/webpack-dev-server/bundle** 运行应用。这会触发一个默认的你不能控制的 *index.html*， 它同样会触发一个允许iFrame中显示重合并的过程。

模块

Webpack allows you to use different module patterns, but "under the hood" they all work the same way. All of them also works straight out of the box.

Webpack 允许你使用不同的模块类型，但是 “底层”必须使用同一种实现。所有的模块都能够开箱即用。

ES6 模块

```
import MyModule from './MyModule.js';
```

CommonJS

```
var MyModule = require('./MyModule.js');
```

AMD

```
define(['./MyModule.js'], function (MyModule) {

});
```

理解文件路径

A module is loaded by filepath. Imagine the following tree structure:

一个模块会按它的文件路径来加载，看一下下面的这个结构：

- /app
  - /modules
    - MyModule.js
  - main.js (entry point)
  - utils.js

Lets open up the *main.js* file and require *app/modules/MyModule.js* in the two most common module patterns:

打开 *main.js* 然后可以通过下面两种方式引入 *app/modules/MyModule.js*

app/main.js

```
// ES6
import MyModule from './modules/MyModule.js';

// CommonJS
var MyModule = require('./modules/MyModule.js');
```

The `./` at the beginning states "relative to the file I am in now".

最开始的 `./` 是 “相对当前文件路径”

Now let us open the *MyModule.js* file and require **app/utils**.

让我们打开 *MyModule.js* 然后引入 **app/utils**：

app/modules/MyModule.js

```
// ES6 相对路径
import utils from './../utils.js';

// ES6 绝对路径
import utils from '/utils.js';

// CommonJS 相对路径
var utils = require('./../utils.js');

// CommonJS 绝对路径
var utils = require('/utils.js');
```

The **relative path** is relative to the current file. The **absolute path** is relative to the entry file, which in this case is *main.js*.

相对路径是相对当前目录。绝对路径是相对入口文件，这个案例中是 *main.js*。

我需要使用文件后缀么？

No, you do not have to use *.js*, but it highlights better what you are requiring. You might have some .js files, and some .jsx files and even images and css can be required by Webpack. It also clearly differs from required node_modules and specific files.

不，你不需要去特意去使用 *.js*，但是如果你引入之后高亮会更好。你可能有一些 .js 文件和一些 .jsx 文件，甚至一些图片和 css 可以用 Webpack 引入，甚至可以直接引入 node_modules 里的代码和特殊文件。

Remember that Webpack is a module bundler! This means you can set it up to load any format you want given there is a loader for it. We'll delve into this topic later on.

记住，Webpack 只是一个模块合并！也就是说你可以设置他去加载任何你写的匹配，只要有一个加载器。我们稍后会继续深入这个话题。

I remember when I saw React the first time around the time it was announced I was skeptical. Particularly mixing some sort of HTML within your code seemed against good conventions. It just felt like a "bad idea"®.

我记得当我第一次看到 React 的时候是十分怀疑的，特别是把 HTML 揉进代码里违背了传统的那些约定，看起来像是一个“坏注意”。

But that's what React and similar approaches are doing. They challenge some of the conventions and replace them with something more palatable. Sometimes a bigger change in thinking is needed for you to move forward as a developer. That's what React did for me. It takes some powerful ideas from the world of functional programming and then builds on top of those.

但是这就是 React，他们挑战了一些约定，然后用这种方式代替了。作为一个开发者，有时候为了能够改变思考方式是你向前走的一个必经之路。这就是 React 对我的影响，它建立在函数式编程上带给了我很多有意思的想法。

基础功能

Before you can understand React and how it changes web development, there are a few things you should know about it. React itself won't be enough. It solves only the problem of views. You still need to complement it with something else. But that's a good thing.

在你能理解 React 是如何改变 Web 开发之前，这里有一点东西你需要知道的。React 它本身是不够丰富的，它只能解决一些视图上的问题，你仍然需要一些东西去帮助它完成事情。

The greatest and worst feature of frameworks is that they sort of cage you in. As long as you are doing what they expect you to do within their boundaries, everything is fine. It is only after you start to reach beyond those boundaries that problems begin to appear. In a library driven approach you aren't as bound. Initially you might not be as fast or efficient but over time as problems become harder, you will have more choices available.

最伟大的和最差劲的框架总是企图给你建立一个笼子，当你在他们划定的地盘里能够完成任何需求，一切都是很好的，但是，当你需要跨出边界的时候，开始有问题了，在一个库驱动方法里你是不被限制的，最初你可能是高效地完成需求，但是伴随时间的推进，问题开始变得严峻起来，你需要更多可行的选择。

JSX 的基础

React provides a component centric approach to frontend development. You will design your application as smaller components, each of which has it own purpose. Taken to the extreme a component may contain its logic, layout and basic styling. To give you an example of JSX:

React 为前端开发提供了一个组件为中心的方法，你可以为你的应用设计一个更小的组件，所有组件有各自的目的，甚至极端来说，一个组件可能包含它自身的逻辑，结构和基本的样式。这里有个 JSX 的例子：

```
...
<TodoItem className='urgent' owner={owner} task='Make a dinner' />
...
```

You can see a couple of basic features of JSX here. Instead of using `class`, we'll use the JavaScript equivalent. In addition we have defined a couple of custom properties in form of `owner` and `task`. `owner` is something that is injected from a variable named `owner` that's within the same scope as our JSX. For `task` we provide a fixed value.

你可以看到一个基本功能的 JSX，我们用 `className` 替代了使用 `class`。此外，我们定义了一些自定义属性 `owner` 和 `task`，`owner` 的值是从 JSX 同环境中一个变量叫 `owner` 注入，而我们为 `task` 提供了一个固定的值。

In practice you would most likely structure this a little differently to fit your data model better. That goes a little beyond basic

React, though.

在实践中，你可以根据你的数据模型结构来稍微调整目录结构会更好，虽然是 React 进阶的知识点了。

We can mix normal JavaScript code within those {}'s. We can use this idea to render a list of `TodoItem`s like this (ES syntax):

我们可以用那些 `{}` 来混合普通 JavaScript 代码，我们可以用这个点像这样去去渲染一个 `TodoItem` 列表（ES 语法）：

```
<ul>{todoItems.map((todoItem, i) =>
    <li key={'todoitem' + i}><TodoItem owner={todoItem.owner} task={todoItem.task} /></li>
)}</ul>
```

You probably noticed something special here. What is that `key` property about? It is something that tells React the exact ordering of your items. If you don't provide unique keys for list items like this, React will warn you as it won't be able to guarantee the correct ordering otherwise.

你可能注意到这里有些特殊的地方，什么是 `key` 属性？这是告诉 React 这个项的准确顺序。如果你不需要像这样需要提供列表项的唯一 Key 的话，否则 React 会警告你这样它没办法保证是正确的顺序。

This has to do with the fact that React actually implements something known as Virtual DOM (VDOM for short) on top of actual DOM. It is a subset of DOM that allows React to optimize its rendering. The primary advantage of this approach is that it allows React to eschew a lot of legacy our good old DOM has gained through years. This is the secret to React's high performance.

需要了解的事实是 React 实际上在真实 DOM 上实现了虚拟 DOM（简称 VDOM），这是一个 DOM 的子集，能够让 React 能够优化它的渲染。这个优化的方法是它让 React 能够避开困恼我们多年的 DOM 性能损耗。这就是 React 高性能的原因。

整体组件

To give you a better idea of what components look like, let's expand our `TodoItem` example into code (ES6 + JSX). I've done this below and will walk you through it:

为了能够让你更加清楚组件长什么样子，让我们拓展 `TodoItem` 例子来感受一下（ES6 + JSX），我已经搞完了，然后我们来过一遍看看：

```
var React = require('react');


module.exports = React.createClass({
    getInitialState() {
        return {
            // 让我们保持追踪看看我们给项点了多少次赞
            likes: 0,
        };
    },
    render() {
        var owner = this.props.owner;
        var task = this.props.task;
        var likes = this.state.likes;

        return <div className='TodoItem'>
            <span className='TodoItem-owner'>{owner}</span>
            <span className='TodoItem-task'>{task}</span>
            <span className='TodoItem-likes'>{likes}</span>
            <span className='TodoItem-like' onClick={this.like}>Like</span>
        </div>;
    },
    like() {
        this.setState({
            likes: this.state.likes + 1
```

```
        });
    },
});
```

You can see some basic features of a React component above. First we create a class for our component. After that we define some initial state for it, then we render and finally we define some custom callbacks for our handlers if they exist. In this case I decided to implement an extra feature, liking. The current implementation just keeps track of the amount of likes per component.

你可以在上面看到一些基础的 React 组件的功能，一开始我们创建了一个组件的类，然后我们在初始化的时候定义一些状态，然后我们渲染，最后我们定制了一些回调。在这个例子中，我决定去实施一个额外的功能，点赞。这个工作只是保持了一个跟踪点赞组件的计数。

In practice you would transmit like amounts to a backend and add some validation there but this is a good starting point for understanding how state works in React.

在实践中，你需要把点赞计数传输给后端，然后添加一些验证，不过目前这个阶段对于理解 State 在 React 中的是如何使用的已经是一个非常好的开端了。

`getInitialState` and `render` are a part of a React component's lifecycle as documented officially. There are additional hooks that allow you to do things like set up adapters for `jQuery` plugins and such.

`getInitialState` 和 `render` 是 React 组件的生命周期官方文档 的一部分。也有额外的能够让你去设置加载 `jQuery` 插件之类的适配器的钩子。

In this example CSS naming has been modeled after Suit CSS conventions as those look clean to me. That's just one way to deal with it.

在例子中的 CSS 的命名是根据 Suit CSS 约定处理后的，这样对我来说看起来非常干净，这只是一种解决方案而已。

处理操作

Let's say we want to modify the owner of our TodoItems. For the sake of simplicity let's expect it's just a string and owner is the same for all TodoItems. Based on this design it would make sense to have an input for owner at our user interface. A naive implementation would look something like this:

让我们来开始修改我们的 TodoItems 的 owner，为了简化这个目的，我们假设 owner 只是一个字符串而且只有一个，基于这个设计，在用户界面里增加一个输入框给用户来修改用户名，改动是这样的：

```
var React = require('react');
var TodoItem = require('./TodoItem.jsx');


module.exports = React.createClass({
    getInitialState() {
        return {
            todoItems: [
                {
                    task: 'Learn React',
                },
                {
                    task: 'Learn Webpack',
                },
                {
                    task: 'Conquer World',
                }
            ],
            owner: 'John Doe',
        };
    },
```

```
    render() {
        var todoItems = this.state.todoItems;
        var owner = this.state.owner;

        return <div>
            <div className='ChangeOwner'>
                <input type='text' defaultValue={owner} onChange={this.updateOwner} />
            </div>

            <div className='TodoItems'>
                <ul>{todoItems.map((todoItem, i) =>
                    <li key={'todoitem' + i}>
                        <TodoItem owner={owner} task={todoItem.task} />
                    </li>
                )}</ul>
            </div>
        </div>;
    },

    updateOwner() {
        this.setState({
            owner: e.target.value,
        });
    },
});
```

We could push `TodoItems` and `ChangeOwner` to separate components but I've kept it all in the same for now. Given React has one way binding by default, we get some extra noise compared to some other setups. React provides ReactLink helper to help deal with this particular case.

我们可以把 `TodoItems` 和 `ChangeOwner` 分离出去，但是我暂时不这么做了。React 默认提供了单向数据绑定，我们可以通过设置来调整，React 提供了 ReactLink 来提供双向数据绑定。

Even though lack of two way binding might sound like a downer, it actually isn't that bad a thing. It makes it easier to reason about the system. You simply have to follow the flow. This idea is highlighted in the Flux architecture. The easiest way to visualize it is to think up an infinite waterfall or a snake eating its tail. That's how the flow in the world of React generally works. Compared to this two way binding feels more chaotic.

尽管双向数据绑定的缺失听起来让人沮丧，但实际上并不是一件坏事，它能够让系统更加轻松的运行，你可能需要跟踪数据流。不过 FB 提出了 Flux 架构，最简单的方式去想出一个无限瀑布流，或者贪吃蛇，就是 React 的世界在做的这个数据流的事情，对比这两种绑定方式让人感觉更加迷茫。

使用一个Mixin

If we wanted to model the code above using a ReactLink, we would end up with something like this:

如果我们想使用 ReactLink的话，就想下面这样：

```
// ReactLink 是一个插件，所以我们需要把它引入。
var React = require('react/addons');

...

module.exports = React.createClass({
    mixins: [React.addons.LinkedStateMixin],

    ...

    render() {
        var todoItems = this.state.todoItems;

        return <div>
            <div className='ChangeOwner'>
                <input type='text' valueLink={this.linkState('owner')} />
            </div>
```

```
            <div className='TodoItems'>
                <ul>{todoItems.map((todoItem, i) =>
                    <li key={'todoitem' + i}>
                        <TodoItem owner={owner} task={todoItem.task} />
                    </li>
                )}</ul>
            </div>
        </div>;
    },
});
```

Now we can skip that `onChange` handler. That `React.addons.LinkedStateMixin` encapsulates the logic. Mixins provide us one way to encapsulate shared concerns such as this into something which can be reused easily.

现在我们可以跳过绑定 `onChange` 事件了， `React.addons.LinkedStateMixin` 封装了逻辑。Mixins 给我们提供了一种可以重复使用的封装方法。

> It would be easy to start expanding the example now. You could for instance provide means to manipulate the contents of the Todo list or start extracting various parts into components of their own. It is up to you to make the app yours. If you are still feeling a bit lost, please read on. This is supposed to be a brief introduction to the topic!
>
> 现在拓展例子已经非常容易了，你现在应该就可以拓展 Todo 列表，或者把各种组件分离，这取决于你。如果你现在还是觉得不够掌握，那么继续阅读，这节只是话题的一个介绍。

测试

If you get serious about the Todo app, I recommend trying Jest out. Getting the initial test run might be a bit challenging but after you learn the basics of the API, it gets a lot simpler. The basic idea is that you instantiate a component with some properties and then query DOM using Jest and finally assert that the values in the UI are what you expect.

如果你觉得这个 Todo 应用需要测试，那么我推荐 Jest，刚开始写测试用例可能是一点挑战但是在你学习一些基础的 API 之后，它会变得更加简单，最基本的是你实例化一个带有一些属性的组件的时候，然后用 Jest 查询 DOM，最后断言界面中的值是你期望的值。

When you go beyond component level, that is where tools such as Selenium come in. You can use standard end to end testing tools on a higher level.

当你站在组件层级之上时，就有了比如类似 Selenium 的组件，你可以在更高层级使用标准的端对端测试工具。

Flux 架构及其变种

As you saw above, it is quite simple to throw together a couple of components and start building an app. You can get quite far with `props` and `state`. Just load up some data over AJAX at `getInitialState` and pass it around. After a while this all might start feeling a bit unwieldy. Why, for instance, my components should have to know something about how to communicate with the backend?

就想你看到的，把一些组件放到一起就可以组成一个应用，你可以用 `props` 和 `state` 做的更多，或者在 `getInitialState` 中使用 AJAX 加载数据然后传给其他组件。这些使用一段时间之后可能会觉得有些笨拙，为什么？实践中，组件们需要知道如何和后端通讯。

This is where Flux architecture and its variants come in. I will start by describing Reflux, a simplified variant of it. You can then work up to understanding Flux in fuller detail once you understand this simplified setup.

所以出现了 Flux 架构和它的一些变种，我会介绍 Reflux 一种非常简单的 Flux 变种，你可以阅读这个 understanding Flux 来全面了解 Flux。

In addition to View Components which we just discussed, Reflux introduces the concepts of Actions and Stores. The flow goes like this: Components -> Actions -> Stores -> Components. Ie. you could have some control in a Component which then triggers some Action which then performs some operation (say PUT) and updates Store state. This state change is then propagated to Components listening to the Store.

并且，Reflux 介绍了一种 Actions 和 Stores 的概念去组成刚才我们讨论的试图组件。整个数据流是这样的：组件 -> Actions -> Stores -> 组件。这样你可以在一个组件中控制触发一些 Action 然后执行一些操作（比如 PUT）然后更新 Store 的状态，状态的更新传播给监听 Store 的那些组件们。

In case of our Todo example we would define basic `TodoActions` like create, update, delete and such. We would also have a `TodoStore`. It would maintain a data structure of a `TodoList`. Our components would then consume that data and display it as appropriate.

在我们的这个 Todo 例子中，我们定义了一个基础的 `TodoActions` 比如创建、更新、删除之类的，我们也有个 `TodoStore`，它是整个 `TodoList` 的数据结构中心，组件们会读取那里的数据，然后适当得展现出来。

As development of Reflux is quite in flux I won't give you a full example in this case. I just wanted to illustrate one possible way to deal with scaling up from bare React. You should explore various options and deepen your understanding of possible architectures. The ideas are quite similar but the devil is in the details as always. There are always drawbacks to consider.

Reflux 的开发和 Flux 差不多，我就不在这里详细讲述了，我只是想说明如何处理从纯 React 扩大的一种可能的方式。你需要去浏览一些相关看法，然后深入理解那些架构。那些想法都差不多，就是细节不太一样，都有各自的一些缺点有待考略。

同构渲染

One of the big features which React provides thanks to its design is so called isomorphic rendering. Back in the day we used to render whole HTML in the backend and provide just that for the client to render. Then we would sprinkle a little JavaScript magic to make things more interactive and so on. After a while the pendulum swung to frontend side. We served minimal amount of HTML to the client and constructed the rest, including routing, using JavaScript entirely on frontend.

React 设计了一种叫做两端渲染的重要功能。过去我们在后端渲染整个 HTML，然后交给客户端去渲染，然后我们加上一点点 JavaScript 做的交互。再后来，这个活交给浏览器做去，后端返回最小的 HTML 给客户端，然后在客户端构建了一整套用 JavaScript 全权控制的系统，包括路由。

The main problems with frontend driven rendering have to do with performance, high dependency on JavaScript (think of the noscript folk!) and poor SEO. With isomorphic rendering you can mitigate these problems effectively. React allows you to prerender HTML at backend side. You can also hydrate some stores with pre-existing data making it possible to skip certain data queries altogether initially! Even web crawlers will be happy as they get some HTML to scrape.

前端驱动渲染的主要问题是性能问题、 JavaScript 的高度依赖（设想那些不能运行 JavaScript 的情况）和 SEO 问题。使用两端渲染，你可以轻松解决这些问题。React 允许你在服务器端预渲染 HTML，你也可以在服务器端预先存储一些数据来跳过初始化的一些数据查询。甚至一些网络爬虫能够轻松地获取到一些 HTML 内容。

This is still partly uncharted territory. Various implementations of Flux still struggle with the concept. I have no doubt we will see stronger solutions in the future, however, as people learn to deal with isomorphism better. That said isomorphic rendering can be considered a nice extra capability to have but it definitely isn't something that's just must have. There are some ways to work around certain issues, such as poor SEO, even without it. It just depends on where you want to put the effort.

这一部分仍然是一个未知的领域（译者注：原文发布于2015年4月），很多 Flux 的实现在这个概念上还没同意，但是毋庸置疑，我们在未来能够看到更加完善的解决方案，无论如何，大家会认识到两端渲染的好处。虽然两端渲染听起来是一个非常好的解决方案，但是它不是必要的。现在已经有很多能够解决这些问题的方案，比如 SEO 的问题，甚至没有用这种解决方案，它只是依赖于你想要如何付出努力。

安装 React JS

`npm install react --save`

There is really nothing more to it. You can now start using React JS in your code.

没什么好讲的，接下来就可以在你的代码中使用 React JS 了。

在代码中使用 ReactJS

component.jsx

```
import React from 'react';

export default class Hello extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}
```

main.js

```
import React from 'react';
import Hello from './component';

main();

function main() {
    React.render(<Hello />, document.getElementById('app'));
}
```

build/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <div id="app"></div>

    <script src="http://localhost:8080/webpack-dev-server.js"></script>
    <script src="bundle.js"></script>
  </body>
</html>
```

转换 JSX

To use the JSX syntax you will need webpack to transform your JavaScript. This is the job of a loader. We'll use Babel as it's nice and has plenty of features.

为了能够使用 JSX 语法，你需要用 Webpack 来转码你的 JavaScript，这是加载器的工作，我们可以使用一个很好用也有很多功能的 Babel。

`npm install babel-loader --save-dev`

Now we have to configure webpack to use this loader.

现在我们需要去配置 Webpack 来使用加载器。

webpack.config.js

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js'),
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.jsx?$/, // 用正则来匹配文件路径，这段意思是匹配 js 或者 jsx
      loader: 'babel' // 加载模块 "babel" 是 "babel-loader" 的缩写
    }]
  }
};

module.exports = config;
```

Webpack will test each path required in your code. In this project we are using ES6 module loader syntax, which means that the require path of `import MyComponent from './Component.jsx';` is `'./Component.jsx'`.

Webpack 会在你的项目中测试所有路径，如果我们项目中使用 ES6 模块加载器语法，比如 `import MyComponent from './Component.jsx';` 是会去匹配 `'./Component.jsx'`。

Run `npm run dev` in the console and refresh the page to see something.

在命令行中运行 `npm run dev`，然后刷新页面就可以看到修改。

You might notice after requiring React JS into your project that the time it takes from a save to a finished rebundle of your application takes more time. In development you ideally want from 200-800 ms rebundle speed, depending on what part of the application you are working on.

你可能注意到在引入 React JS 到你的项目之后，给你的应用重新合并会花费太多的时间。在开发环境中，最理想的是编译最多 200 到 800 毫秒的速度，取决于你在开发的应用。

在开发环境中使用压缩文件

Instead of making Webpack go through React JS and all its dependencies, you can override the behavior in development.

为了不让 Webpack 去遍历 React JS 及其所有依赖，你可以在开发中重写它的行为。

webpack.config.js

```
var path = require('path');
var node_modules = path.resolve(__dirname, 'node_modules');
var pathToReact = path.resolve(node_modules, 'react/dist/react.min.js');

config = {
    entry: ['webpack/hot/dev-server', path.resolve(__dirname, 'app/main.js')],
    resolve: {
        alias: {
          'react': pathToReact
        }
    },
    output: {
        path: path.resolve(__dirname, 'build'),
        filename: 'bundle.js',
    },
    module: {
        loaders: [{
            test: /\.jsx?$/,
            loader: 'babel'
        }],
        noParse: [pathToReact]
    }
};

module.exports = config;
```

We do two things in this configuration:

我们在配置中做了两件事：

1. Whenever "react" is required in the code it will fetch the minified React JS file instead of going to *node_modules*

2. Whenever Webpack tries to parse the minified file, we stop it, as it is not necessary

1. 每当 "react" 在代码中被引入，它会使用压缩后的 React JS 文件，而不是到 *node_modules* 中找。
2. 每当 Webpack 尝试去解析那个压缩后的文件，我们阻止它，因为这不必要。

Take a look at Optimizing development for more information on this.

可以到 优化开发 看到更多这方面的信息。

If you come to JavaScript from other programming languages you are familiar with types. You have types in JavaScript too, but you do not have to specify these types when declaring variables, receiving arguments etc. This is one of the things that makes JavaScript great, but at the same time not so great.

如果你是从其他你更熟悉类型的编程语言转到 JavaScript 的，那么，你也可以在 JavaScript 中使用类型，不过你不需要在声明变量或者接收参数的时候指定类型。这可以让你的代码更加优雅，但是不是最优雅的。

Specifically when working on very large projects with many developers type checking gives stability to your project, much like a good test does. So using **Flow** is definitely not a requirement. It is for developers who depends on type checking as more of a routine and for the before mentioned large projects with many developers. Webpack makes it easy to include **Flow** in your workflow.

具体来说，当你在一个大项目中和很多开发者一起工作的时候，类型检查对你的项目来讲是保证了稳定性，就像一个很好的测试做的事情。所以使用 **Flow** 当然不是必要条件。对于那些需要和很多开发者一起做大项目的开发者来说类型检测是一种非常寻常的事。Webpack 引入 **Flow** 之后会非常方便。

安装 flow

- 还没尝试：）
- 尝试一下 "flowcheck-loader"? https://www.npmjs.com/package/flowcheck-loader (还没用过：））
- https://tryflow.org/

Webpack allows you to load CSS like you load any other code. What strategy you choose is up to you, but you can do everything from loading all your css in the main entry point file to one css file for each component.

Webpack允许像加载任何代码一样加载 CSS。你可以选择你所需要的方式，但是你可以为每个组件把所有你的 CSS 加载到入口主文件中来做任何事情。

Loading CSS requires the **css-loader** and the **style-loader**. They have two different jobs. The **css-loader** will go through the CSS file and find `url()` expressions and resolve them. The **style-loader** will insert the raw css into a style tag on your page.

加载 CSS 需要 **css-loader** 和 **style-loader**，他们做两件不同的事情，**css-loader**会遍历 CSS 文件，然后找到 `url()` 表达式然后处理他们，**style-loader** 会把原来的 CSS 代码插入页面中的一个 style 标签中。

准备加载 CSS

Install the two loaders: `npm install css-loader style-loader --save-dev`.

安装这两个加载器：`npm install css-loader style-loader --save-dev`

In the *webpack.config.js* file you can add the following loader configuration:

你可以把下面的加载器配置加到 *Webpack.config.js* 文件中。

webpack.config.js

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js')
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.jsx$/,
      loader: 'jsx'
    }, {
      test: /\.css$/, // Only .css files
      loader: 'style!css' // Run both loaders
    }]
  }
};

module.exports = config;
```

加载 CSS 文件

Loading a CSS file is a simple as loading any file:

加载一个 CSS 文件就和加载其他文件一样简单：

main.js

```
import './main.css';
// Other code
```

Component.jsx

```
import './Component.css';
import React from 'react';

export default React.createClass({
  render: function () {
    return <h1>Hello world!</h1>
  }
});
```

Note! You can of course do this with both CommonJS and AMD.

注意！ 你也可以在 CommonJS 和 AMD 中做同样的事情。

CSS 加载策略

Depending on your application you might consider three main strategies. In addition to this you should consider including some of your basic CSS inlined with the initial payload (index.html). This will set the structure and maybe a loader while the rest of your application is downloading and executing.

根据你的应用，你可能会考略三种策略。另外，你需要考虑把一些基础的 CSS 内联到初始容器中（index.html），这样设置的结构能够在应用下载和执行的时候加载剩下的应用。

所有合并成一个

In your main entry point, e.g. `app/main.js` you can load up your entire CSS for the whole project:

在你的主入口文件中个，比如 `app/main.js` 你可以为整个项目加载所有的 CSS：

app/main.js

```
import './project-styles.css';
// 其他 JS 代码
```

The CSS is included in the application bundle and does not need to download.

CSS 就完全包含在合并的应用中，再也不需要重新下载。

懒加载

If you take advantage of lazy loading by having multiple entry points to your application, you can include specific CSS for each of those entry points:

如果你想发挥应用中多重入口文件的优势，你可以在每个入口点包含各自的 CSS：

app/main.js

```
import './style.css';
// 其他 JS 代码
```

app/entryA/main.js

```
import './style.css';
// 其他 JS 代码
```

app/entryB/main.js

```
import './style.css';
// 其他 JS 代码
```

You divide your modules by folders and include both CSS and JavaScript files in those folders. Again, the imported CSS is included in each entry bundle when running in production.

你把你的模块用文件夹分离，每个文件夹有各自的 CSS 和 JavaScript 文件。再次，当应用发布的时候，导入的 CSS 已经加载到每个入口文件中。

具体的组件

With this strategy you create a CSS file for each component. It is common to namespace the CSS classes with the component name, thus avoiding some class of one component interfering with the class of an other.

你可以根据这个策略为每个组件创建 CSS 文件，可以让组件名和 CSS 中的 class 使用一个命名空间，来避免一个组件中的一些 class 干扰到另外一些组件的 class。

app/components/MyComponent.css

```
.MyComponent-wrapper {
  background-color: #EEE;
}
```

app/components/MyComponent.jsx

```
import './MyComponent.css';
import React from 'react';

export default React.createClass({
  render: function () {
    return (
      <div className="MyComponent-wrapper">
        <h1>Hello world</h1>
      </div>
    )
  }
});
```

使用内联样式取代 **CSS** 文件

With "React Native" you do not use stylesheets at all, you only use the *style-attribute*. By defining your CSS as objects. Depending on your project, you might consider this as your CSS strategy.

在 “React Native” 中你不再需要使用任何 CSS 文件，你只需要使用 *style* 属性，可以把你的 CSS 定义成一个对象，那样就可以根据你的项目重新来考略你的 CSS 策略。

app/components/MyComponent.jsx

```
import React from 'react';

var style = {
```

```
  backgroundColor: '#EEE'
};

export default React.createClass({
  render: function () {
    return (
      <div style={style}>
        <h1>Hello world</h1>
      </div>
    )
  }
});
```

When **webpack-dev-server** is running with Automatic browser refresh the CSS will also update, but a bit differently. When you do a change to a CSS file the style tag belonging to that file will be updated with the new content... without a refresh!

当 **Webpack-dev-server** 在 浏览器自动刷新 下运行的时候，CSS 也会自动更新，不过有点不同的是，当你改变了一个 CSS 文件，属于那个文件的标签会更新新的内容但不会刷新。

If you want to use compiled CSS, there are two loaders available for you. The **less-loader** and the **sass-loader**. Depending on your preference, this is how you set it up.

如果你想使用编译 CSS，这里有两种可用的加载器：**less-loader** 和 **sass-loader**，看你喜欢哪种。下面是如何设置。

安装和设置加载器

`npm install less-loader` or `npm install sass-loader`.

`npm install less-loader` 或者 `npm install sass-loader`.

webpack.config.js

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js')
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.jsx$/,
      loader: 'jsx'
    },

    // LESS
    {
      test: /\.less$/,
      loader: 'style!css!less'
    },

    // SASS
    {
      test: /\.scss$/,
      loader: 'style!css!sass'
    }]
  }
};
```

LESS 和 SASS 中的 imports 怎么办?

If you import one LESS/SASS file from an other, use the exact same pattern as anywhere else. Webpack will dig into these files and figure out the dependencies.

如果你从另外一个文件中导入一个 LESS/SASS 文件，像其他地方一样使用准确的路径，Webpack 会找出那些文件，然后识别里面的依赖。

```
@import "./variables.less";
```

You can also load LESS files directly from your node_modules directory.

你也可以直接从你的 node_modules 文件夹中加载 LESS 文件。

```
$import "~bootstrap/less/bootstrap";
```

Until HTTP/2 is here you want to avoid setting up too many HTTP requests when your application is loading. Depending on the browser you have a set number of requests that can run in parallel. If you load a lot of images in your CSS it is possible to automatically inline these images as BASE64 strings to lower the number of requests required. This can be based on the size of the image. There is a balance of size of download and number of downloads that you have to figure out for your project, and Webpack makes that balance easy to adjust.

直到 HTTP/2 你才能在应用加载的时候避免设置太多 HTTP 请求。根据浏览器不同你必须设置你的并行请求数，如果你在你的 CSS 中加载了太多图片的话，可以自动把这些图片转成 BASE64 字符串然后内联到 CSS 里来降低必要的请求数，这个方法取决与你的图片大小。你需要为你的应用平衡下载的大小和下载的数量，不过 Webpack 可以让这个平衡十分轻松适应。

安装 url-loader

`npm install url-loader --save-dev` will install the loader that can convert resolved paths as BASE64 strings. As mentioned in other sections of this cookbook Webpack will resolve "url()" statements in your CSS as any other require or import statements. This means that if we test on image file extensions for this loader we can run them through it.

`npm install url-loader --save-dev` 来安装加载器，它会把需要转换的路径变成 BASE64 字符串，在其他的 Webpack 书中提到的这方面会把你 CSS 中的 “url()” 像其他 require 或者 import 来处理。意味着如果我们可以通过它来处理我们的图片文件。

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js')
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.jsx$/,
      loader: 'jsx'
    }, {
      test: /\.(png|jpg)$/,
      loader: 'url?limit=25000'
    }]
  }
};
```

The limit is an argument passed to the url-loader. It tells it that images that er 25KB or smaller in size will be converted to a BASE64 string and included in the CSS file where it is defined.

url-loader 传入的 limit 参数是告诉它图片如果不大于 25KB 的话要自动在它从属的 css 文件中转成 BASE64 字符串。

Fonts can be really difficult to get right. First of all we have typically 4 different formats, but only one of them will be used by the respective browser. You do not want to inline all 4 formats, as that will just bloat your CSS file and in no way be an optimization.

字体实在是非常难引入正确，首先，通常我们有 4 种不一样的格式，但是只有其中一种会被对应的浏览器使用到。你肯定不会想引入全部四种格式，这样只会让 CSS 文件更加膨胀，然后又没办法优化。

选择一种格式

Depending on your project you might be able to get away with one font format. If you exclude Opera Mini, all browsers support the .woff and .svg format. The thing is that fonts can look a little bit different in the different formats, on the different browsers. So try out .woff and .svg and choose the one that looks the best in all browsers.

取决与你的项目，你可能可以选择出一种字体格式，如果你不考略 Opera Mini，所有的浏览器都支持 .woff 和 .svg 格式。问题是不同格式下在各种浏览器下字体看起来会有一点点不同。所以测试 .woff 和 .svg，然后找出能够在所有浏览器中看起来最好的那个。

There are probably other strategies here too, so please share by creating an issue or pull request.

如果有其他更好的策略，那请通过创建 issue 或者提 PullRequest 来分享。

实践

You do this exactly like you do when inlining images.

就像内联图片一样来内联字体。

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js')
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.jsx$/,
      loader: 'jsx'
    }, {
      test: /\.woff$/,
      loader: 'url?limit=100000'
    }]
  }
};
```

Just make sure you have a limit above the size of the fonts, or they will of course not be inlined.

There are two things you want to do preparing for a production build.

这里有两件事你需要为生产发布做准备。

1. Configure a script to run in your package.json file
2. Create a production config

1. 配置你的 package.json 里的脚本
2. 创建一个生产的配置

创建脚本

We have already used *package.json* to create the `npm run dev` script. Now let us set up `npm run deploy`.

我们已经使用过 *package.json* 来创建 `npm run dev` 的脚本，现在让我们设置 `npm run deploy`。

```
{
  "name": "my-project",
  "version": "0.0.0",
  "description": "My awesome project!",
  "main": "app/main.js",
  "scripts": {
    "dev": "webpack-dev-server --devtool eval --progress --colors --hot --content-base build",
    "deploy": "NODE_ENV=production webpack -p --config webpack.production.config.js"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^1.4.13",
    "webpack-dev-server": "^1.6.6"
  },
  "dependencies": {}
}
```

As you can see we are just running webpack with the production argument and pointing to a different configuration file. We also use the environment variable "production" to allow our required modules to do their optimizations. Lets us create the config file now.

正如你所见，我们只是用生产参数运行 Webpack 来指向另一个配置文件。我们也使用了环境变量 “production” 来让我们的模块自动去优化。让我们开始来创建配置文件。

创建生产配置

So there really is not much difference in creating the dev and production versions of your webpack config. You basically point to a different output path and there are no workflow configurations or optimizations. What you also want to bring into this configuration is cache handling.

可以看到，其实生产环境的配置和开发的配置没有太大的不同，主要的不同是指向了一个不同的输出路径，然后也没有了 workflow 的配置和优化，可以看到新加入到配置里的是处理缓存的配置。

```
var path = require('path');
var node_modules_dir = path.resolve(__dirname, 'node_modules');

var config = {
  entry: path.resolve(__dirname, 'app/main.js'),
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
```

```
        test: /\.js$/,

        // There is not need to run the loader through
        // vendors
        // 这里再也不需通过任何第三方来加载
        exclude: [node_modules_dir],
        loader: 'babel'
      }]
    }
  };

  module.exports = config;
```

发布

Run `npm run deploy` in the root of the project. Webpack will now run in production mode. It does some optimizations on its own, but also React JS will do its optimizations. Look into caching for even more production configuration.

在项目根目录处运行 `npm run deploy`，Webpack 现在会运行生产模式，他会自动做一些优化，不过，React Js 也会做自己的优化。可以深入了解缓存处理来做更多的生产配置。

Lets have a look at the simplest setup you can create for your application. Use a single bundle when:

让我们看一下为应用创建的最简单的配置，只有在下面的情况下才使用单入口模式：

- You have a small application
- You will rarely update the application
- You are not too concerned about perceived initial loading time

- 应用很小
- 很少会更新应用
- 你不太关心初始加载时间

webpack.production.config.js

```
var path = require('path');
var config = {
  entry: path.resolve(__dirname, 'app/main.js'),
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.js$/,
      loader: 'babel'
    }]
  }
};

module.exports = config;
```

When your application is depending on other libraries, especially large ones like React JS, you should consider splitting those dependencies into its own vendors bundle. This will allow you to do updates to your application, without requiring the users to download the vendors bundle again. Use this strategy when:

当你的应用依赖其他库尤其是像 React JS 这种大型库的时候，你需要考虑把这些依赖分离出去，这样就能够让用户在你更新应用之后不需要再次下载第三方文件。当满足下面几个情况的时候你就需要这么做了：

- When your vendors reaches a certain percentage of your total app bundle. Like 20% and up
- You will do quite a few updates to your application
- You are not too concerned about perceived initial loading time, but you do have returning users and care about optimizing the experience when you do updates to the application
- Users are on mobile

- 当你的第三方的体积达到整个应用的 20% 或者更高的时候。
- 更新应用的时候只会更新很小的一部分
- 你没有那么关注初始加载时间，不过关注优化那些回访用户在你更新应用之后的体验。
- 有手机用户。

webpack.production.config.js

```
var path = require('path');
var webpack = require('webpack');
var node_modules_dir = path.resolve(__dirname, 'node_modules');

var config = {
  entry: {
    app: path.resolve(__dirname, 'app/main.js'),

    // Since react is installed as a node module, node_modules/react,
    // we can point to it directly, just like require('react');
    // 当 React 作为一个 node 模块安装的时候，
    // 我们可以直接指向它，就比如 require('react')
    vendors: ['react']
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'app.js'
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: [node_modules_dir],
      loader: 'babel'
    }]
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin('vendors', 'vendors.js')
  ]
};

module.exports = config;
```

This configuration will create two files in the `dist/` folder. **app.js** and **vendors.js**.

这些配置会在 `dist/` 文件夹下创建两个文件：**app.js** 和 **vendors.js**。

重要的事情！

Remember to add both files to your HTML file, or you will get the error: `Uncaught ReferenceError: webpackJsonp is not defined`.

记住要把这些文件都加入到你的 HTML 代码中，不然你会得到一个错误：`Uncaught ReferenceError: webpackJsonp is not`

`defined`。

Maybe you are building an application that has multiple urls. An example of this would be a solution where you have two, or more, different URLs responding with different pages. Maybe you have one user page and one admin page. They both share a lot of code, but you do not want to load all the admin stuff for normal users. That is a good scenario for using multiple entry points. A list of use cases could be:

你的应用可能有多个路径， 就是应用中有两个或者多个 URL 相应不同的页面，这里就是提供这样的解决方案。可能你有一个普通用户页和一个管理员页，他们共享了很多代码，但是不想在普通用户页中加载所有管理员页的代码，所以好方案是使用多重入口。使用缘由有下面几条：

- You have an application with multiple isolated user experiences, but they share a lot of code
- You have a mobile version using less components
- You have a typical user/admin application where you do not want to load all the admin code for a normal user

- 你的应用有多种不同的用户体验，但是他们共享了很多代码。
- 你有一个使用更少组件的手机版本
- 你的应用是典型的权限控制，你不想为普通用户加载所有管理用户的代码。

Let us create an example with a mobile experience using less components:

让我们创建一个使用更少组件的手机页面的例子：

webpack.production.config.js

```
var path = require('path');
var webpack = require('webpack');
var node_modules_dir = path.resolve(__dirname, 'node_modules');

var config = {
  entry: {
    app: path.resolve(__dirname, 'app/main.js'),
    mobile: path.resolve(__dirname, 'app/mobile.js'),
    vendors: ['react'] // 其他库
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js' // 注意我们使用了变量
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: [node_modules_dir],
      loader: 'babel'
    }]
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin('vendors', 'vendors.js')
  ]
};

module.exports = config;
```

This configuration will create three files in the `dist/` folder. **app.js**, **mobile.js** and **vendors.js**. Most of the code in the **mobile.js** file also exists in **app.js**, but that is what we want. We will never load **app.js** and **mobile.js** on the same page.

这个配置会在 `dist/` 文件夹下创建三个文件：**app.js**、**mobile.js**和**vendors.js**，大部分的代码在**mobile.js**文件中，也有一部分在 **app.js** 中，不过这是我们需要的，我们不会在同一个页面中同时加载 **app.js** 和 **mobile.js**。

It is also possible to lazy load entry points. This means that you load parts of your application as they are requested. A typical scenario for this would be that your users only visits specific parts of the application. And an example of that would be twitter.com. You do not always visit your profile page, so why load the code for that? Here is a summary of requirements:

Webpack 也可以实现懒加载入口文件，意味着应用的一部分只在需要的时候加载，一个典型的例子是用户只有访问一些应用特定的部分，典型的例子是 Twitter.com，你不会一直访问你的个人页，所以为什么要加载那部分的代码？这里有个主要的要求：

- You have a relatively big application where users can visit different parts of it
- You do care a lot about initial render time

- 你有一个相对比较大的应用，可以让用户可以访问应用的不同部分。
- 你非常关注初始渲染时间

webpack.production.config.js

```
var path = require('path');
var webpack = require('webpack');
var node_modules_dir = path.resolve(__dirname, 'node_modules');

var config = {
  entry: {
    app: path.resolve(__dirname, 'app/main.js'),
    vendors: ['react']
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'app.js'
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: [node_modules_dir],
      loader: 'babel'
    }]
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin('vendors', 'vendors.js')
  ]
};

module.exports = config;
```

So we are pretty much back where we started with a split application and vendors bundle. You do not really define your lazy dependencies in a configuration, Webpack automatically understands them when analyzing your code. So let us see how we would lazy load a **profile page**:

所以我们把应用和第三方分离是一件非常漂亮的事，你不需要在配置中设置懒加载依赖，Webpack 会自动理解他们，然后分析你的代码。所以让我们看看我们是如何加载一个 个人信息页：

main.js (使用 ES6 语法)

```
import React from 'react';
import Feed from './Feed.js';

class App extends React.Component {
  constructor() {
    this.state = { currentComponent: Feed };
  }
  openProfile() {
    require.ensure([], () => {
      var Profile = require('./Profile.js');
      this.setState({
        currentComponent: Profile
```

```
        });
      });
    }
    render() {
     return (
        return <div>{this.state.currentComponent()}</div>
      );
    }
  }
  React.render(<App/>, document.body);
```

So this is just an example. You would probably hook this up to a router, but the important part is using `require.ensure`.

这只是一个例子，你需要把这些写入到一个路由中个，不过重要的事情是使用了 `require.ensure`。

What is the array on the first argument?: If you try to lazy load a chunk that depends on an other lazy loaded chunk you can set it as a dependency in the array. Just type in the path to the chunk. E.g. `['./FunnyButton.js']`

第一个数组参数是什么？：如果你尝试去懒加载一段由另一个懒加载的代码加载的代码的话，把它作为依赖写在数组里，就把路径写进去，比如 `['./FunnyButton.js']`

So the great thing about React JS is that it runs on the server too. But that does not mean you can just create any app and run it on the server. You have to make some decisions on the architecture. The reason is that even though React JS and the components run on the server, you might be having dependencies in those components that does not run on the server.

React Js 最伟大的地方是它也可以运行在服务端，不过这不意味着你可以创建任何一个应用然后运行在服务端，你需要做一些决策和架构。原因是哪怕 React JS 和一些组件可以在服务端运行，但还是有一些组件中的依赖不能在服务端运行。

注入状态

One of the most important decisions you make is to inject the state of your application through the top component. This basically means that your components does not have any external dependencies at all. All they need to know comes through this injected state.

一个重要的事情是应用需要通过顶层组件把状态注入，这意味着你的组件没有了任何的外部依赖，他们只能通过注入的状态来获取信息。

This cookbook is not about isomorphic apps, but let us take a look at an example. We will not use ES6 syntax here because Node JS does not support it yet.

这本小书不是主要讲同构渲染的应用，不过让我们来看一下例子，我们这次不使用 ES6 语法了，因为 Node JS 还不完全支持。

main.js (client)

```
var React = require('react');
var AppState = require('./client/AppState.js');
var App = require('./App.js');

React.render(<App state={AppState}/>, document.body);
```

router.js (server)

```
var React = require('react');
var App = require('./App.js');
var AppState = require('./server/AppState.js');
var index = '<!DOCTYPE html><html><head></head><body>{{component}}</body></html>';

app.get('/', function (req, res) {
  var componentHtml = React.renderToString(App({state: AppState}));
  var html = index.replace('{{component}}', componentHtml);
  res.type('html');
  res.send(html);
});
```

So this was a very naive and simple way of showing it, but what you should notice here is that we use the same **App.js** file on the client and server, but we have two different ways of producing the state.

所以这是一个非常初级且简单的例子来展示它，不过你需要注意的是我们在客户端和服务端使用了同一个 **App.js**，但是我们需要两种方式来提供状态。

> TODO: what to discuss in intro? Maybe summarize what makes a pro. Like using hot loader, caching etc. hm hm... maybe look at it after the sub pages are done, cherry on the top :-)

We talked about how you could use the minified versions of your dependencies in development to make the rebundling go as fast as possible. Let us look at a small helper you can implement to make this a bit easier to handle.

之前介绍了如何在开发中使用依赖的压缩版本来让合并尽可能加速，让我们看一下这个小的例子来让你更加轻松去处理：

webpack.config.js

```
var webpack = require('webpack');
var path = require('path');
var node_modules_dir = path.join(__dirname, 'node_modules');

var deps = [
  'react/dist/react.min.js',
  'react-router/dist/react-router.min.js',
  'moment/min/moment.min.js',
  'underscore/underscore-min.js',
];

var config = {
  entry: ['webpack/hot/dev-server', './app/main.js'],
  output: {
    path: path.resolve(__dirname, './build'),
    filename: 'bundle.js'
  },
  resolve: {
    alias: {}
  },
  module: {
    noParse: [],
    loaders: []
  }
};

// Run through deps and extract the first part of the path,
// as that is what you use to require the actual node modules
// in your code. Then use the complete path to point to the correct
// file and make sure webpack does not try to parse it
// 通过在第一部分路径的依赖和解压
// 就是你像引用 node 模块一样引入到你的代码中
// 然后使用完整路径指向当前文件，然后确认 Webpack 不会尝试去解析它

deps.forEach(function (dep) {
  var depPath = path.resolve(node_modules_dir, dep);
  config.resolve.alias[dep.split(path.sep)[0]] = depPath;
  config.module.noParse.push(depPath);
});

module.exports = config;
```

Not all modules include a minified distributed version of the lib, but most do. Especially with large libraries like React JS you will get a significant improvement.

不是所有的模块需要一个压缩的版本，不过大多数需要，尤其是像 React JS 这种大型库，之后你会有明显的提升。

把 React 暴露到全局中

You might be using distributed versions that requires React JS on the global scope. To fix that you can install the expose-loader by `npm install expose-loader --save-dev` and set up the following config, focusing on the *module* property:

你可能在全局中使用了一个压缩版本的 React，为了修复你可以安装这个暴露全局加载器 `npm install expose-loader --save-dev`，然后像下面这样配置，注意 *module* 属性：

```
var webpack = require('webpack');
var path = require('path');
var node_modules_dir = path.join(__dirname, 'node_modules');
```

```
var deps = [
  'react/dist/react.min.js',
  'react-router/dist/react-router.min.js',
  'moment/min/moment.min.js',
  'underscore/underscore-min.js',
];

var config = {
  entry: ['webpack/hot/dev-server', './app/main.js'],
  output: {
    path: path.resolve(__dirname, './build'),
    filename: 'bundle.js'
  },
  resolve: {
    alias: {}
  },
  module: {
    noParse: [],

    // Use the expose loader to expose the minified React JS
    // distribution. For example react-router requires this
    // 使用暴露全局加载器来暴露压缩版的 React JS，比如 react-router 需要这个。
    loaders: [{
      test: path.resolve(node_modules_dir, deps[0]),
      loader: "expose?React"
    }]
  }
};

deps.forEach(function (dep) {
  var depPath = path.resolve(node_modules_dir, dep);
  config.resolve.alias[dep.split(path.sep)[0]] = depPath;
  config.module.noParse.push(depPath);
});

module.exports = config;
```

So this part is just freakin' awesome. With React JS and the react-hot-loader you can change the class code of your component and see the instances update live in the DOM, without loosing their state! This is pretty much exactly how CSS updates behave, only that it is your components.

所以这个章节就是他妈的屌。使用 React JS 和 react-hot-loader 可以让你去改变组件中的 class 代码，然后可以在 DOM 上看到实时更新了实例，没有修改他们的状态！看起来就像 CSS 更新一样，不过是换成了组件。

设置

This setup requires that you use the **webpack-dev-server** as introduced in earlier chapters. Now we just have to install the loader with `npm install react-hot-loader --save-dev`, do a small config change:

这个设置需要你使用前面章节中介绍的 **webpack-dev-server**，现在我们需要去安装加载器 `npm install react-hot-loader --save-dev`，然后做一点配置：

```
var webpack = require('webpack');
var path = require('path');

var config = {
  entry: ['webpack/hot/dev-server', './app/main.js'],
  output: {
    path: path.resolve(__dirname, './build'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [{
      test: /\.js$/,

      // Use the property "loaders" instead of "loader" and
      // add "react-hot" in front of your existing "jsx" loader
      // 使用 "loaders" 属性代替 "loader"
      // 然后在 "jsx" 加载器之前添加 "react-hot"
      loaders: ['react-hot', 'babel']
    }]
  }
};

module.exports = config;
```

And you will also need a small snippet of code in your main entry file. In the example above that would be the *main.js* file located in the `app/` folder.

同时你也需要在你的主入口文件做一些修改，例子中，在 `app/` 文件夹中的 *main.js* 像下面那样修改：

app/main.js

```
// You probably already bring in your main root component,
// maybe it is your component using react-router
// 你可能已经把你的根组件引入了
// 组件可能用了 react-router
var RootComponent = require('./RootComponent.jsx');

// When you render it, assign it to a variable
// 当你渲染它的时候，让它赋值给一个变量
var rootInstance = React.render(RootComponent(), document.body);

// Then just copy and paste this part at the bottom of
// the file
// 然后在文件的最底部复制粘帖
if (module.hot) {
  require('react-hot-loader/Injection').RootInstanceProvider.injectProvider({
    getRootInstances: function () {
      // Help React Hot Loader figure out the root component instances on the page:
      // 帮助 React Hot Loader 识别出页面中的根组件
```

```
        return [rootInstance];
      }
    });
  }
```

It is that simple. Render a component to the DOM and make a code change on the class of that component. It will render itself again, keeping the existing state. Cool?

就是这么简单，在 DOM 中渲染一个组件，然后修改一些组件中的代码，它会自动渲染，却保存了已经存在了的状态，屌不屌？

Read more about the react-hot-loader.

可以到 react-hot-loader 了解更多。

When users hit the URL of your application they will need to download different assets. CSS, JavaScript, HTML, images and fonts. The great thing about Webpack is that you can stop thinking how you should download all these assets. You can do it through JavaScript.

当用户输入你应用的地址的时候，他们需要去下载不同的资源，比如 CSS、JavaScript、HTML、图片和字体。不过 Webpack 做了一件事情，让你不用去考虑如何不用下载全部资源。

> OccurenceOrderPlugin

如何让生产输出附上哈希值？

- Use `[hash]`. Example: `'assets/bundle.[hash].js'`
- 使用 `[hash]`。比如：`'assets/bundle.[hash].js'`

The benefit of this is that this will force the client to reload the file. There is more information about `[hash]` at the long term caching section of the official documentation.

这个的好处是能够让客户端强制重新加载这个文件，可以在 the long term caching 了解更多关于 `[hash]`，

> Is it possible to change the hash only if bundle changed? 有可能只有合并文件变化了才会修改哈希值么？

我可以使用哪种匹配器?

- `{ test: /\.js$/, loader: 'babel-loader' }` - Matches just .js
- `{ test: /\.(js|jsx)$/, loader: 'babel-loader' }` - Matches both js and jsx
- Generally put it's just a JavaScript regex so standard tricks apply

- `{ test: /\.js$/, loader: 'babel-loader' }` - 只匹配 .js
- `{ test: /\.(js|jsx)$/, loader: 'babel-loader' }` - 匹配 js 和 jsx
- 一般来说它就是一段 JavaScript 的正则，所以按照标准来即可

- Explain how webpack thinks chunks and not files
- What are files to load? And what does webpack create for you? And how?

- 解释 Webpack 如何理解 chunks 而不是文件。

- 加载什么文件？Webpack 会创建怎么样的？如何？

Webpack can be handy for packaging your library for general consumption. You can use it to output UMD, a format that's compatible with various module loaders (CommonJS, AMD) and globals.

Webpack 可以非常方便地打包和生成你的库，你可以用它输出 UMD，一种可以兼容很多模块加载器（CommonJS、AMD）和全局变量的格式。

如何把库输出成 UMD?

Especially if you are creating a library, it can be useful to output an UMD version of your library. This can be achieved using the following snippet:

尤其是如果你创建了一个库，那么输出一个 UMD 版本是非常有用的，就像下面的片段一样实现：

```
output: {
    path: './dist',
    filename: 'mylibrary.js',
    libraryTarget: 'umd',
    library: 'MyLibrary',
},
```

In order to avoid bundling big dependencies like React, you'll want to use a configuration like this in addition:

为了避免去合并类似 React 的大型依赖，你可以使用下面这样的设置：

```
externals: {
    react: 'react',
    'react/addons': 'react'
},
```

如何输出压缩版?

Here's the basic idea:

这里是一个简单的方案：

```
output: {
    path: './dist',
    filename: 'awesomemular.min.js',
    libraryTarget: 'umd',
    library: 'Awesomemular',
},
plugins: [
    new webpack.optimize.UglifyJsPlugin({
        compress: {
            warnings: false
        },
    }),
]
```